	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorio de Computacion Salas A y B

Profesor(a): Rene Adrian Davila Perez

Asignatura: Programacion orientada a objetos

Grupo: 01

No de practica(s): 78

Integrante(s): 322001549, 322006829, 322125533, 425131152, 322103351

No de lista o brigada: 07

Semestre: 2026-1

Fecha de entrega: viernes 17 de octubre del 2025

Observaciones:

Calificacion:

Índice

1. Introducción

En la mayoría de los lenguajes de programación orientado a objetos cada clase que se crea está implícitamente heredado en la clase *Object*, por ello se pueden comportar como objetos. En la herencia es posible que nuevos objetos tengan propiedades de otros ya existentes, para ello, existe una clase llamada súper clase o clase base.

Si una clase hereda algo de una clase base, se conocerá como subclase o clase derivada, esta va a heredar todas las propiedades de la clase base, así como métodos e incluso agregar sus propias características. En relación a lo anterior, el polimorfismo ayuda a compartir mismos métodos en diferentes objetos, donde cada uno pueden implementarlos a su manera, también se puede aplicar para datos, lo que permite a un método evaluar y ser aplicado de forma indistinta. Todos los objetos en Java pueden considerarse polimórficos, pues todos se comportan como objetos de su propio tipo y como objetos de la clase *Object*.

1.1. Planteamiento del problema

Se realiza una interfaz en lenguaje Java que calcule el área y perímetro para un rectángulo, círculo y triángulo rectángulo. La interfaz deberá presentar una sección para que el usuario seleccione la figura deseada, además, debe existir un apartado que permita ingresar la base y altura de la figura, de tal modo que, en el caso del rectángulo y triángulo rectángulo, se permita calcular las características ya mencionadas (para el círculo solo será necesario ingresar el radio).

1.2. Motivación

La reutilización de código permite mantener una arquitectura ordenada y simplificada, si el polimorfismo permite mantener métodos de manera particular, la práctica contribuye al conocimiento de cómo estos principios crean programas más flexibles, modulares y fáciles de mantener.

1.3. Objetivos

- Completar la implementación vista en clase.
- Demostrar cómo una interfaz puede definir el comportamiento común entre clases para el cálculo del área y perímetro en las figuras mencionadas anteriormente.
- Inclusión de herencia y polimorfismo de la clase abstracta *Figura* a las clases; *Circulo*, *Rectangulo* y *TrianguloRectangulo*.
- Empaquetar los códigos del programa dentro del *Full Qualified Name mx.unam.fi.poo.p78*.

2. Marco Teórico

2.1. Clase

Una clase en Java define un nuevo tipo de datos que puede incluir campos (variables) y métodos para definir el comportamiento de los objetos creados a partir de la clase: [1]

```

class ClassName {
    // Fields
    dataType fieldName;

    // Constructor
    ClassName(parameters) {
        // Initialize fields
    }

    // Methods
    returnType methodName(parameters) {
        // Method body
    }
}
```

De donde:

- ClassName: El nombre de la clase.
- fieldName: Variables que contienen el estado de la clase.
- Constructor: Método especial utilizado para inicializar objetos.
- methodName: Funciones definidas dentro de la clase para realizar acciones.

2.2. Objeto

Un objeto es una instancia de una clase. Se crea utilizando la palabra clave new seguida del constructor de la clase: [1]

```


ClassName objectName = new ClassName(arguments);
```

De donde:

- objectName: El nombre del objeto.
- arguments: Valores pasados al constructor para inicializar el objeto

A continuación un ejemplo incluyendo ambos conceptos:

```java

class Car {
    // Fields
    String color;
    int year;

    // Constructor
    Car(String color, int year) {
        this.color = color;
        this.year = year;
    }

    // Method
    void displayInfo() {
        System.out.println("Car color: " + color + ", Year: " +
    year);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object
        Car myCar = new Car("Red", 2020);
        myCar.displayInfo(); // Output: Car color: Red, Year: 2020
    }
}
```

En este ejemplo, la clase Car tiene los campos color y year, un constructor para inicializar estos campos, y un método displayInfo() para mostrar los detalles del coche. Se crea un objeto myCar utilizando la clase Car.

2.3. Constructor

En Java, un constructor es un método especial utilizado para inicializar objetos. A diferencia de los métodos normales, los constructores se invocan cuando se crea una instancia de una clase. Tienen el mismo nombre que la clase y no tienen tipo de retorno. Los constructores son esenciales para establecer los valores iniciales de los atributos del objeto y prepararlo para su uso.[2]

Tipos de constructores:

1. Constructor por defecto: El compilador Java proporciona automáticamente un constructor por defecto si no se definen constructores explícitamente en la clase. Inicializa el objeto con valores por defecto.
2. Constructor sin argumentos: Un constructor sin argumentos es definido explícitamente por el programador y no recibe ningún parámetro. Es similar al constructor por defecto, pero puede incluir código de inicialización personalizado.[2]
3. Constructor parametrizado: Un constructor parametrizado acepta argumentos para inicializar un objeto con valores específicos. Esto permite una inicialización más flexible y controlada de los atributos de los objetos.[2]
4. Copiar Constructor: Un constructor de copia se utiliza para crear un objeto nuevo como copia de un objeto existente. Java no proporciona un constructor de copia por defecto; sin embargo, puede implementarse manualmente.[2]

A continuación sus sintaxis de constructor por defecto:

```
class ClassName {
    // Constructor
    ClassName() {
        // Initialization code
    }

    // Parameterized Constructor
    ClassName(dataType parameter1, dataType parameter2) {
        // Initialization code using parameters
    }

    // Copy Constructor
    ClassName(ClassName obj) {
        // Initialization code to copy attributes
    }
}
```

Ejemplo de constructor:

```
public class Car {
    String model;
    int year;

    // Default Constructor
    public Car() {
        model = "Unknown";
        year = 0;
    }

    public static void main(String[] args) {
        Car car = new Car();
        System.out.println("Model: " + car.model + ", Year: " +
    car.year);
    }
}
```

En este ejemplo, la clase Car tiene un constructor por defecto que inicializa los atributos model y year con valores por defecto. Cuando se crea un objeto Car, se le asignan estos valores por defecto. [2]

Ejemplo de constructor parametrizado:

```
public class Car {
    String model;
    int year;

    // Parameterized Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public static void main(String[] args) {
        Car car = new Car("Toyota", 2021);
        System.out.println("Model: " + car.model + ", Year: " +
    car.year);
    }
}
```

Ejemplo de sobrecarga de constructores:

```java
public class Car {
    String model;
    int year;

    // No-Argument Constructor
    public Car() {
        model = "Unknown";
        year = 0;
    }

    // Parameterized Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("Honda", 2022);
        System.out.println("Car1 -> Model: " + car1.model + ",
    Year: " + car1.year);
        System.out.println("Car2 -> Model: " + car2.model + ",
    Year: " + car2.year);
    }
}
```

Este ejemplo demuestra la sobrecarga de constructores en la clase Car, donde se definen tanto un constructor sin argumentos como un constructor parametrizado. Esto permite crear objetos con valores predeterminados o específicos.[2]

2.4. Métodos, parámetros y argumentos

2.4.1. Métodos

Los métodos son bloques de código que se utilizan para realizar tareas específicas. Cada método tiene un nombre que lo identifica y puede recibir una serie de parámetros, los cuales son variables que proporcionan información necesaria para que el método realice su tarea. Estos parámetros actúan como entrada para el método, permitiéndole operar con los valores recibidos.[3]

Dentro del cuerpo del método, se pueden realizar diversas operaciones, como cálculos matemáticos, manipulación de datos o incluso llamadas a otros métodos. Una vez que el método ha completado sus operaciones, puede devolver un resultado utilizando la palabra clave return seguida del valor que se desea devolver.[3]

La capacidad de reutilizar código es una de las ventajas clave de los métodos en Java. Al definir un método una vez, podemos llamarlo en diferentes partes de nuestro programa, evitando la repetición innecesaria de código. Esto no solo simplifica nuestro código, sino que también facilita su mantenimiento y permite una mayor modularidad en nuestra aplicación.[3]

```java
public tipo_de_dato_devuelto nombre_metodo(tipo_de_dato parametro1
    , tipo_de_dato parametro2) {
    // codigo a ejecutar
    return resultado;
}
```

2.4.2. Parámetros

Son las variables que se definen en la declaración de un método y sirven para recibir información externa. Actúan como entradas que el método necesita para ejecutar sus instrucciones.[3]

```
public void saludar(String nombre) {
    System.out.println("Hola, " + nombre + "!    Cmo    est s?");
}
```

En el método denominado “saludar” se recibe un parámetro de tipo String llamado “nombre”. Dentro del cuerpo del método, se imprime una cadena de saludo que incluye el valor del parámetro “nombre”. Al llamar a este método, debemos proporcionar un argumento de tipo String que se asignará al parámetro “nombre”.

2.4.3. Argumentos

Son los valores concretos que se pasan a un método cuando este es llamado. Estos valores corresponden a los parámetros definidos en el método. Una vez que hemos declarado un método, podemos llamarlo desde otro lugar de nuestro código. La llamada de un método se realiza utilizando su nombre seguido de paréntesis, y se pueden pasar los argumentos correspondientes si es necesario.[3]

```
int resultado = sumar(5, 3); // Llamada a un metodo con retorno
```

```
public int sumar(int a, int b) {
    return a + b;
}
```

2.5. Encapsulamiento

La encapsulación es un principio fundamental de la programación orientada a objetos (POO) en Java que consiste en agrupar los datos (variables) y los métodos (funciones) que operan sobre los datos en una sola unidad, conocida como clase. Restringe el acceso directo a algunos de los componentes de un objeto y puede evitar la modificación accidental de datos. [4]

La finalidad de la encapsulación es la siguiente:

- Ocultar datos: La encapsulación permite ocultar al exterior la representación interna de un objeto. Sólo se exponen los detalles necesarios a través de una interfaz pública.
- Mayor flexibilidad: Al controlar el acceso a los campos de una clase, puedes cambiar la implementación interna sin afectar al código externo que utiliza la clase.
- Mantenimiento mejorado: La encapsulación ayuda a mantener el código manteniendo los campos privados y proporcionando métodos getter y setter públicos para modificar y ver los campos.

Para lograr la encapsulación en Java:

1. Declara las variables de clase como private.
2. Proporciona los métodos getter y setter de public para acceder y actualizar el valor de una variable privada.

A continuación un ejemplo:

```java
public class Student {
    private String name;
    private int age;

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}
```

En este ejemplo, la clase Student tiene los campos privados name y age. Se proporcionan métodos públicos getter y setter para acceder a estos campos y modificarlos. El configurador de age incluye una comprobación de validación básica.

A continuación un segundo ejemplo con validación:

```java
public class BankAccount {
    private double balance;

    // Getter method for balance
    public double getBalance() {
        return balance;
    }

    // Setter method for balance with validation
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

La clase BankAccount encapsula el campo balance. Proporciona los métodos deposit y withdraw para modificar la balanza, garantizando que sólo se realizan operaciones válidas.

2.6. Herencia

La herencia es uno de los mecanismos fundamentales de la programación orientada a objetos, por medio del cual una clase se construye a partir de otra. Una de sus funciones más importantes es proveer el polimorfismo. Relaciona las clases de manera jerárquica; una clase padre, superclase o clase base sobre otras clases hijas, subclases o clase derivada. [5].

Los descendientes de una clase heredan todos los atributos y métodos que sus ascendientes hayan especificado como heredables, además de crear los suyos propios. En Java, sólo se permite la herencia simple, o dicho de otra manera, la jerarquía de clases tiene estructura de árbol. El punto más alto de la jerarquía es la clase *Object* de la cual derivan todas las demás clases. Para especificar la superclase se realiza con la palabra *extends*; si no se especifica, se hereda de Object. [5].

2.6.1. Definición de superclase

Comienza definiendo una clase base, también llamada superclase, que contiene los atributos y métodos comunes que se desean compartir con las clases derivadas [6].

```
1
2  public class Animal {
3      String nombre;
4      public void comer() {
5         System.out.println("El animal esta comiendo.");
6      }
7  }
```

2.6.2. Creación de clase derivada

Luego, crea una nueva clase que herede de la clase base usando la palabra clave extends.

```
1
2  public class Perro extends Animal {
3      public void ladrar() {
4          System.out.println("El perro esta ladrando.");
5      }
6  }
```

Con lo anterior, es posible crear objetos de la clase derivada, permitiendo el acceso a los atributos y métodos en clase base o derivada.

```
1
2  public class Main {
3      public static void main(String[] args) {
4          Perro miPerro = new Perro();
5          miPerro.nombre = "Bobby";
6          miPerro.comer();  // Metodo de la clase base
7          miPerro.ladrar(); // Metodo de la clase derivada
8      }
9  }
```

En este ejemplo, la clase Perro hereda de la clase Animal, por lo que un objeto de la clase Perro también puede acceder a los métodos y atributos de la clase Animal. La herencia permite compartir y reutilizar funcionalidades comunes [6].

2.7. Polimorfismo

El polimorfismo es la capacidad de que un mismo mensaje funcione con diferentes objetos. Es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo concreto de objetos sobre el que se trabaja, el método opera sobre un conjunto de posibles objetos compatibles.[5].

Para crear una clase base, se deben definir los métodos que desean que las clases derivadas implementen de forma específica, por ejemplo:

```
1
2  public interface Animal {
3     void hacerSonido();
4  }
```

Posteriormente se crean clases concretas que implementen la interfaz o hereden de su clase base. Cada base debe proporcionar su propia implementación del método definido anteriormente.

```
1
2  public class Perro implements Animal {
3      public void hacerSonido() {
4          System.out.println("El perro ladra.");
5      }
6  }
7
8  public class Gato implements Animal {
9      public void hacerSonido() {
10         System.out.println("El gato maulla.");
11     }
12 }
```

Ahora es posible crear objetos de clases derivadas y tratarlos como si se tratase de objetos de la clase base, por lo que es posible llamar un método común, en el Main de esta manera:

```
1
2  public class Main {
3      public static void main(String[] args) {
4          Animal miMascota = new Perro();
5          miMascota.hacerSonido(); // Llama al metodo del perro
6
7          miMascota = new Gato();
8          miMascota.hacerSonido(); // Llama al metodo del gato
9      }
10 }
```

En este ejemplo, se crean los objetos objetos Perro y Gato, asignados a una referencia de tipo Animal y se llama al método *hacerSonido()* sin necesidad de la implementación real de cada clase. Esto es el polimorfismo: objetos de diferentes clases responden al mismo mensaje (hacerSonido()) de manera específica para cada uno [6]

2.8. Paquetes

Un paquete (package) en Java es un espacio de nombres (namespace) que organiza un conjunto de clases, interfaces, enumeraciones y anotaciones relacionadas. Actúa como una carpeta lógica que agrupa elementos con funcionalidad similar. [7]

Para crear un paquete, debemos utilizar la declaración del paquete agregándola como la primera línea de código en un archivo . [8]

Se recomienda encarecidamente colocar cada nuevo tipo en un paquete. Si definimos tipos y no los colocamos en un paquete, se incluirán en el paquete predeterminado o sin nombre. Usar paquetes predeterminados tiene algunas desventajas: [8]

- Perdemos los beneficios de tener una estructura de paquete y no podemos tener subpaquetes.
- No podemos importar los tipos del paquete predeterminado desde otros paquetes.
- Los ámbitos de acceso protegido y privado del paquete no tendrían sentido.

Por ejemplo, para crear un paquete a partir de www.baeldung.com , invirtámoslo:

```
com.baeldung
```

Comencemos definiendo una clase TodoItem en un subpaquete llamado dominio :

```
package com.baeldung.packages.domain;

public class TodoItem {
    private Long id;
    private String description;

    // standard getters and setters
}
```

Al compilar nuestras clases empaquetadas, debemos recordar la estructura de directorios. Empezando por la carpeta de origen, debemos indicarle a javac dónde encontrar nuestros archivos. [8]

Primero necesitamos compilar nuestra clase TodoItem porque nuestra clase TodoList depende de ella.Comencemos abriendo una línea de comando o terminal y navegando a nuestro directorio de origen. Ahora, compilemos nuestra clase com.baeldung.packages.domain.TodoItem :

```
> javac com/baeldung/packages/domain/TodoItem.java
```

Ahora que nuestra clase TodoItem está compilada, podemos compilar nuestras clases TodoList y TodoApp :

```
>javac -classpath . com/baeldung/packages/*.java
```

Ejecutemos nuestra aplicación usando el nombre completo de nuestra clase TodoApp :

```
>java com.baeldung.packages.TodoApp
```

2.9. Graphics2D

La sección sobre Graphics2D del libro Introduction to Computer Graphics explica cómo Java implementa los gráficos bidimensionales mediante la clase Graphics2D, la cual amplía las funciones básicas de la clase Graphics. Todo dibujo en Java se realiza dentro del método paintComponent(Graphics g), y ese objeto Graphics puede convertirse en Graphics2D para acceder a funciones avanzadas como suavizado, transformaciones y modelado jerárquico. Por ejemplo, una de las primeras instrucciones que se muestran en la página es la activación del suavizado (antialiasing), que mejora la calidad de las líneas: [9]

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
```

A partir de ahí, se describe que Graphics2D trabaja con objetos geométricos llamados shapes, como Line2D, Rectangle2D, Ellipse2D o Path2D, que permiten crear figuras con coordenadas reales. Dichas formas pueden dibujarse con draw() o rellenarse con fill(), y el estilo se define mediante setColor() o setPaint(), pudiendo usar colores sólidos, gradientes o patrones, como se ve en el ejemplo PaintDemo.java. [9]

Una de las secciones más importantes trata sobre las transformaciones afines, que modifican el sistema de coordenadas para desplazar, rotar o escalar las figuras. En la documentación se presentan de la siguiente manera:

```
g2.scale(sx,sy) -escala mediante un factor de escala horizontal sx
     y un factor de escala vertical sy.
g2.rotate(r)   gira   el   ngulo   r alrededor del origen, medido en
    radianes. Un   ngulo   positivo gira el eje x positivo en la
    direcci n del eje y positivo.
g2.rotate(r,x,y)   gira   el   ngulo   r alrededor del punto (x, y).
g2.translate(dx,dy)   traslada   dx horizontalmente y dy
    verticalmente.
g2.shear(sx,sy)  aplica   una fuerza cortante horizontal sx y una
    fuerza cortante vertical sy. (Normalmente, una de las fuerzas
    cortantes es 0, lo que da como resultado una fuerza cortante
    horizontal o vertical pura).
```

En Java, una transformación se representa con un objeto de la clase AffineTransform. Una transformación general puede construirse así:

```
AffineTransform trns = new AffineTransform(a,b,c,d,e,f);
```

Y transforma un punto (x, y) en un nuevo punto (x, y) según las ecuaciones:

```
 x    = a*x + c*y + e
 y    = b*x + d*y + f
```

Estas operaciones permiten manipular figuras de manera precisa sin recalcular coordenadas, y son fundamentales para el modelado jerárquico, donde una figura compleja se forma a partir de subfiguras independientes que conservan su propio sistema de coordenadas. En los ejemplos del libro, como HierarchicalModeling2D.java, se muestra cómo un carrito con ruedas y molinos de viento se compone de piezas más simples, cada una transformada y dibujada de forma local, creando una escena coherente y animada. También se menciona SceneGraphAPI2D.java, que implementa el mismo principio pero usando una estructura de grafo de escena para organizar jerárquicamente los objetos. [9]

3. Desarrollo

3.1. Descripción de la Implementación

Para esta práctica se retomó el código base provisto por el profesor, el cual incluía la clase abstracta `Figura` y la interfaz gráfica implementada con `Swing` (`MainApp`, `PanelDibujo` y `NumericTextField`). Nuestra tarea consistió en completar la implementación de las subclases `Circulo`, `Rectangulo` y `TrianguloRectangulo`, aplicando **herencia y polimorfismo**.

En cada clase se sobreescribieron los métodos `area()` y `perimetro()` mediante la palabra clave `@Override`, se implementaron sus respectivos *setters* y *getters* para encapsular los atributos y se incluyó la instrucción de empaquetado `package mx.unam.fi.poo.p78;` al inicio de cada archivo.

3.2. Solución aplicada: Herencia y Polimorfismo

La clase `Figura` fue proporcionada por el profesor y define el comportamiento común que todas las figuras deben tener. A partir de esta clase abstracta, en las figuras ya dadas cómo archivos, se sobreescribió el metodo`area()` y `perimetro()`, aprovechando el concepto de **herencia**.

3.3. Subclases implementadas

Círculo

En la clase `Circulo` se encapsuló el atributo `radio` y se implementaron los métodos sobrescritos `area()` y `perimetro()` con las fórmulas correspondientes. Además, se agregaron los métodos `setRadio()` y `getRadio()` para controlar el acceso al atributo, evitando valores negativos.

```
1  package mx.unam.fi.poo.p78;
2
3  public class Circulo extends Figura {
4      private double radio;
5
6      public Circulo(double radio) { setRadio(radio); }
7
8      public void setRadio(double radio) { this.radio = Math.max(0,
       radio); }
9      public double getRadio() { return radio; }
10
11     @Override
12     public double area() { return Math.PI * radio * radio; }
13
14     @Override
15     public double perimetro() { return 2 * Math.PI * radio; }
16 }
```

Rectángulo

La clase `Rectangulo` incluye los atributos `ancho` y `alto`, ambos encapsulados con sus respectivos métodos `setters` y `getters`. Se implementaron las fórmulas del área y perímetro, asegurando que los valores negativos se reemplacen por cero.

```java
package mx.unam.fi.poo.p78;

public class Rectangulo extends Figura {
    private double ancho, alto;

    public Rectangulo(double ancho, double alto) {
        setAncho(ancho); setAlto(alto);
    }

    public void setAncho(double v) { this.ancho = Math.max(0, v); }
    public void setAlto(double v)  { this.alto  = Math.max(0, v); }

    public double getAncho() { return ancho; }
    public double getAlto()  { return alto;  }

    @Override
    public double area() { return ancho * alto; }

    @Override
    public double perimetro() { return 2 * (ancho + alto); }
}
```

Triángulo Rectángulo

En la clase `TrianguloRectangulo` se trabajó con los atributos `base` y `altura`, que también fueron encapsulados con sus *getters* y *setters*. El método `area()` calcula el valor usando la fórmula $(base \times altura)/2$, y el método `perimetro()` utiliza la función `Math.hypot()` para obtener la hipotenusa.

```java
package mx.unam.fi.poo.p78;

public class TrianguloRectangulo extends Figura {
    private double base, altura;

    public TrianguloRectangulo(double base, double altura) {
        setBase(base); setAltura(altura);
    }

    public void setBase(double v)   { this.base = Math.max(0, v); }
    public void setAltura(double v) { this.altura = Math.max(0, v); }

    public double getBase()   { return base;   }
    public double getAltura() { return altura; }

    @Override
    public double area() { return (base * altura) / 2.0; }

    @Override
    public double perimetro() {
        return base + altura + Math.hypot(base, altura);
    }
}
```

3.4. Integración con la interfaz gráfica

La interfaz gráfica, desarrollada previamente por el profesor, ya contenía la lógica para recibir los valores desde los campos numéricos y mostrar los resultados. Nuestra implementación permitió que,

al seleccionar una figura, la aplicación construyera el objeto correspondiente (`Circulo`, `Rectangulo` o `TrianguloRectangulo`), invocando sus métodos `area()` y `perimetro()` para mostrar los cálculos en pantalla.

3.5. Empaquetado

Todas las clases fueron incluidas dentro del paquete `mx.unam.fi.poo.p78`, tal como se solicitó en las instrucciones de la práctica, agregando al inicio de cada archivo:

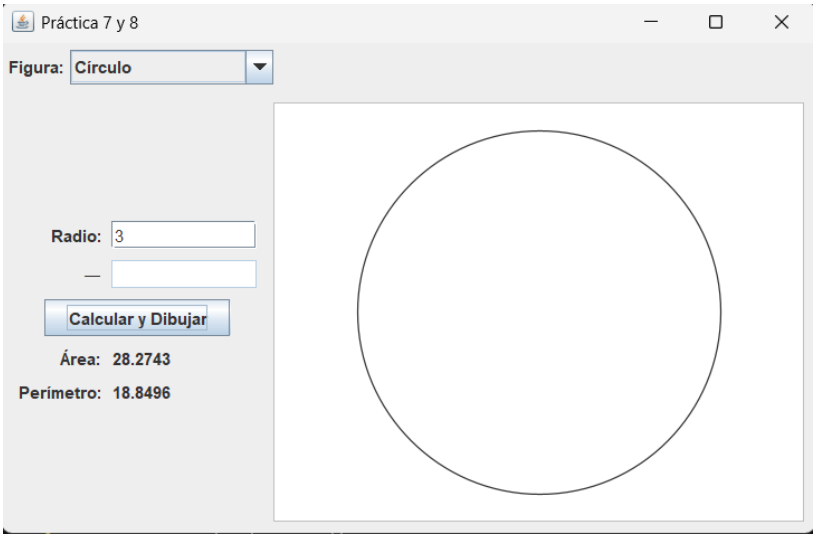
```
1 package mx.unam.fi.poo.p78;
```

4. Resultados

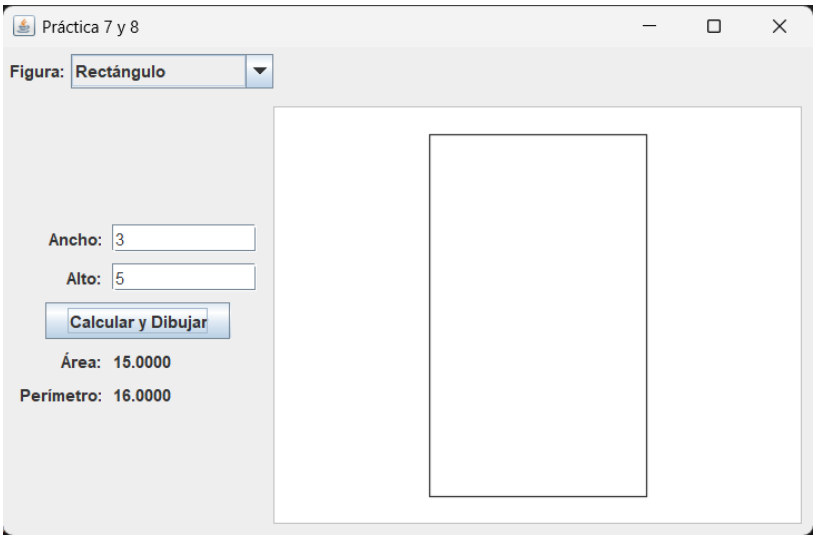
Para probar que el código funcionará de manera correcta después de incluir la herencia y polimorfismo de las clases abstractas, además de haber aplicado el empaquetamiento y encapsulamiento, se ejecutó y compiló de la siguiente manera:

```
PS C:\Users\52551\JavaPruebaXDD> javac -d out mx/unam/fi/poo/p78/*.java
PS C:\Users\52551\JavaPruebaXDD> java -cp out mx.unam.fi.poo.p78.MainApp
```

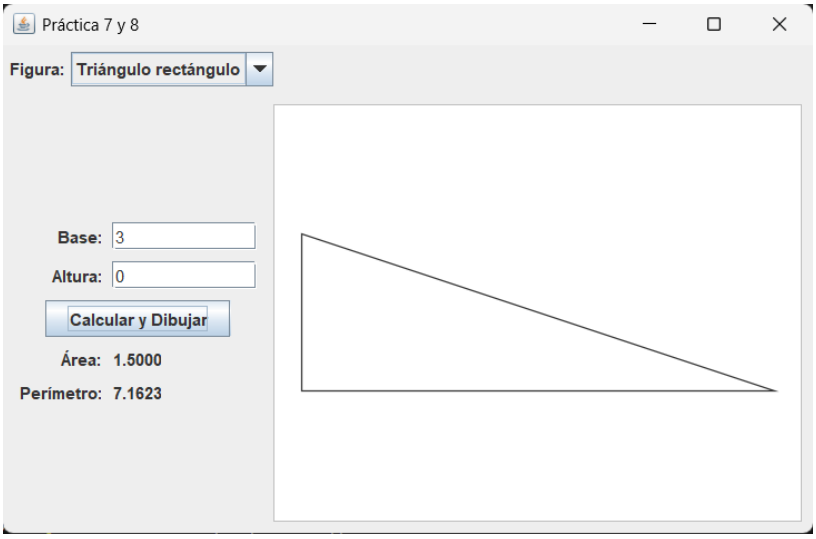
Esto debido a que marcaba errores en la terminal al momento de compilar, lo que hace es compilar los .java y va a guardar los .class en una carpeta de salida (out) y al momento de ejecutar, como lo que se ejecuta es el .class del MainApp.java, se llama desde la carpeta de salida añadiendo lo que está empaquetado y el archivo a compilar.



Podemos observar que al usar la parte que utiliza a la **clase abstracta círculo**, pide el radio que es lo único que se necesita para conocer el área y perímetro de la figura y retorna los resultados correctos de las medidas correspondientes.



Al observar la parte de la **clase abstracta rectángulo** también podemos confirmar que se ejecuta de manera correcta ya que pide el ancho y alto de la figura, calcula correctamente el área y el perímetro y realiza el dibujo de la figura de manera proporcional, no en su tamaño real.



Con la parte de la **clase abstracta triángulo** también se ejecuta de manera correcta, aunque podemos observar un caso particular al colocar algún valor, en este caso en la altura, podemos observar que al colocar 0 nos calcula un área y un perímetro diferentes de cero esto debido a que en la siguiente parte del código indica que si alguno de los valores dados es 0 lo tome como 1 lo que explica el porque de este comportamiento.

```
1         if (p1 <= 0) p1 = 1.0;
2         if (t2.isEnabled() && p2 <= 0) p2 = 1.0;
```

5. Conclusiones

La práctica realizada permitió alcanzar de manera integral los objetivos planteados, fortaleciendo la comprensión y aplicación de los principios fundamentales de la programación orientada a objetos. En primer lugar, se completó exitosamente la implementación de las clases Circulo, *Rectangulo* y *TrianguloRectangulo*, tomando como base la clase abstracta Figura proporcionada. Esta estructura permitió definir un comportamiento común y reutilizable para el cálculo del área y perímetro, cumpliendo así con el objetivo de demostrar cómo una interfaz o clase abstracta puede servir como guía para estandarizar funcionalidades en diferentes clases derivadas.

La herencia fue aplicada correctamente para extender la funcionalidad general de Figura a cada subclase, mientras que el polimorfismo facilitó el uso uniforme de los métodos **area()** y **perimetro()** independientemente del tipo específico de figura. Esta estrategia de diseño no solo mejora la legibilidad y mantenibilidad del código, sino que también fomenta su escalabilidad, permitiendo agregar nuevas figuras en el futuro con mínima modificación a la estructura existente.

Por otro lado, se implementaron correctamente los métodos **getters** y **setters** para cada atributo relevante, garantizando la **encapsulación** de datos y evitando valores inconsistentes o negativos que pudieran afectar los resultados de los cálculos. Además, la integración con la interfaz gráfica preexistente demostró que la lógica interna de las clases funciona adecuadamente, ya que la selección de una figura y la introducción de sus dimensiones permitieron obtener y mostrar en tiempo real el área y el perímetro correspondientes.

Finalmente, se cumplió con el empaquetado del código dentro del Full Qualified Name mx.unam.fi.poo.p78, siguiendo buenas prácticas de organización y modularidad en proyectos Java. Este empaquetado asegura una estructura clara y profesional, facilitando la reutilización y la integración en sistemas más amplios.

En conclusión, esta práctica no solo permitió cumplir los objetivos establecidos, sino que también reforzó de manera práctica conceptos esenciales como abstracción, herencia, polimorfismo, **encapsulación** y **modularidad**, preparando una base sólida para el desarrollo de aplicaciones orientadas a objetos más complejas. Además, la experiencia adquirida contribuye a comprender la importancia de diseñar programas bien estructurados, mantenibles y fácilmente extensibles.

Referencias

[1] DataCamp. (2025) Clases y objetos en java. Accedido: 16 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/classes-and-objects

[2] ——. (2025) Java constructors. Accedido: 16 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/constructors

[3] A. Barragán. (2023) Introducción a java: Métodos, parámetros y argumentos. Publicado: 24 de octubre de 2023; Accedido: 16 de septiembre de 2025. [Online]. Available: https://openwebinars.net/blog/introduccion-a-java-metodos-parametros-y-argumentos/

[4] DataCamp. (2023) Encapsulation in java. Accedido: 30 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/encapsulation

[5] J. B. Bermúdez, *Programación Orientada a Objetos en Java*, Documento académico en línea, Escuela Técnica Superior de Ingenieros en Sistemas Informáticos, Madrid, España, 2013, curso 2013-2014, Máster en Ingeniería Web (MIW). [Online]. Available: https://www.etsisi.upm.es/sites/default/files/curso$_2$013$_1$4/$MASTER/MIW.JEE.POOJ.pdf$

[6] J. L. Blasco, "Introducción a poo en java: Herencia y polimorfismo," 2023, publicado: 17 de noviembre de 2023, Accedido: 2025-10-08. [Online]. Available: https://openwebinars.net/blog/introduccion-a-poo-en-java-herencia-y-polimorfismo/

[7] Oracle. (2025) What is a package? Accedido: 30 de septiembre de 2025. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/concepts/package.html

[8] Baeldung. (2024) Guide to java packages. Accedido: 30 de septiembre de 2025. [Online]. Available: https://www.baeldung.com/java-packages

[9] D. J. Eck. (2025) Java graphics2d. Accedido: 13-octubre-2025. [Online]. Available: